January 2001

Resource Sharing Through Query Process Migration

Arnaud Sahuguet
University of Pennsylvania

Val Tannen
University of Pennsylvania, val@cis.upenn.edu

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-01-10.

Resource Sharing Through Query Process Migration

Abstract

In this paper we focus on systems whose goal is the distribution or exchange of resources such as software packages, scientific data, or multimedia files. To accomplish this goal efficiently, distributed systems find and retrieve resources using combinations of techniques such as brokering, proxying, caching, publish/subscribe, advertising, chaining, referral, recruiting, etc. The resulting architectures are complex and scale with difficulty.

We propose two ideas that combine to make such systems much more scalable. One is to encapsulate queries into processes and to use process migration as the basic primitive for cooperation between sites. With this, a generic *installer* can run at each site, implementing any and all of the techniques mentioned above. The other idea is to describe resource access as "delayed" queries embedded in metadata. Standard distributed database techniques can be applied to the metadata with the additional side-effect of spawning appropriate query processes.

We describe a *distributed query language* that can be obtained by adding our process manipulation primitives to any query language. We discuss installation strategies and we present an algorithm for the site-generic installer on which such a language is based. We also show how to apply query rewriting techniques that allow the installer to perform optimizations. Moreover, we give algorithms that insure that the global amount of installation associated with a given resource retrieval task is bounded.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-01-10.

Resource Sharing Through Query Process Migration

Arnaud Sahuguet Val Tannen

University of Pennsylvania

Abstract

In this paper we focus on systems whose goal is the distribution or exchange of resources such as software packages, scientific data, or multimedia files. To accomplish this goal efficiently, distributed systems find and retrieve resources using combinations of techniques such as brokering, proxying, caching, publish/subscribe, advertising, chaining, referral, recruiting, etc. The resulting architectures are complex and scale with difficulty.

We propose two ideas that combine to make such systems much more scalable. One is to encapsulate queries into processes and to use process migration as the basic primitive for cooperation between sites. With this, a generic *installer* can run at each site, implementing any and all of the techniques mentioned above. The other idea is to describe resource access as "delayed" queries embedded in metadata. Standard distributed database techniques can be applied to to the metadata with the additional side-effect of spawning appropriate query processes.

We describe a *distributed query language* that can be obtained by adding our process manipulation primitives to any query language. We discuss installation strategies and we present an algorithm for the site-generic installer on which such a language is based. We also show how to apply query rewriting techniques that allow the installer to perform optimizations. Moreover, we give algorithms that insure that the global amount of installation associated with a given resource retrieval task is bounded.

Technical Report MS-CIS-01-10

1 Introduction

Resource sharing in software systems is not new. What is new and intriguing is doing it on a large scale in the context of wide-access systems relying on carefully limited cooperation from participants. Looking beyond the hype[1] we aim to discuss here the technological underpinnings that of some flavors of resource sharing in its current incarnation.

We focus on distributed systems that share data and require some cooperation toward query processing. Specifically, we will make the following assumptions:

1. The main purpose of these systems is to *share data*, both in the form of *metadata* and of *"payload"*.
2. In order to facilitate *shared query processing* the sites communicate metadata and *capabilities*.
3. Payloads are large enough that optimizing their processing is worthwhile. Same for some of the metadata *catalogs*.
4. Most queries require the *integration* of metadata from multiple sites.
5. The systems foster and exploit *redundancy* in data and capabilities.
6. Site availability is *variable* both in performance and reliability.
7. The ability to *scale* is essential.

Some of these assumptions suggest techniques such as resolving queries with the participation of "intermediaries", i.e., sites that play the roles of **brokers** and **proxies**. Others suggest widespread use of **caching** and **publish/subscribe**, in fact *fostering* data redundancy. In addition to exploit redundancy in capabilities we would need flexible combinations [14] of **data-shipping** and **query-shipping**) and strategies such as **chaining**, **referral**, and **recruiting**. All in all, many different functionalities. Experience has shown that hard-coding different such functionalities at different sites results in unwieldy systems that are unlikely to scale.

At the other extreme, there is much to be said about the scalability of *peer-to-peer* architectures. We wish in fact to exploit such flexibility and still to combine it with widespread use of techniques like brokering, proxying, etc. Thus, we naturally come to the conclusion that all sites should have the ability to perform various *roles*: caching, publish to subscribers, broker or act as proxies. This leads to another assumption:

8. Participating sites run sandboxed copies of the same software, performing (as we shall see) query installation[2] migration and execution.

This is realized here by designing a *distributed query language*; in fact, a set of primitives for process manipulation that can be added to *any* query language. Thus, instead of hard-coded, we make the behavior of each site *programmable*. The software running at each site can be understood as a copy of the interpreter of the language.

The building bricks of the language are *query processes*. The language borrows ideas from mobile process calculi [32] especially the concept of query process *migration*. Our most important methodological device is the separation of *installation* and *migration* of query processes from their *execution*. Installation is like laying down pipes between the site that originated the query and the sites that have the data, using migration to make use of intermediary sites along the way. Once the pipes are laid down, we can "turn on the faucets" (execution). The data flows and it is processed into the eventual answer.

[1] Eg., Gnutella, Groove and especially Napster have made headlines recently.
[2] Our notion of installation is more general than the one from [28] and [11]. See Section 2.

To some extent, execution follows established experience. The data is streamed through pipelined operators (one does not need all the data to start computing) using channels between sites. A new feature of our language is the ability to mix *delayed* queries inside metadata. A process spawning feature can then "activate" these delayed queries.

Most of our effort is focused on installation. Because there may be many layers of sites between a query site and the data sites, it is not practical to assume globally complete and timely knowledge of the system at any individual site. Indeed, this knowledge cannot be built into the system because of variability and the scale of the system precludes sites looking around aimlessly to discover resources (see discussion about Gnutella in Section 2). Instead, we assume that each site has a *neighborhood* consisting of sites it works with. These in turn work with others, etc., eventually reaching the sites with the desired data. Sites **advertise** their metadata and capabilities to others in their neighborhood. To deal with the system's attempts to improve by setting up caches and subscriptions, the metadata and the capabilities are advertised as **leases**, i.e., they expire after a period of time.

The installation strategies that we investigate attempt to minimize site interaction. For this reason we reject protocols that need to maintain a state in multiple sites during multiple rounds of messages. By making full use of query process migration, our approach is to identify a few basic steps that each site implements and then show how these plus migration can be used to play all the roles mentioned above. Other benefits of choosing a small set of basic mechanisms are reliability of the implementation and the ability to reason formally about the behavior of the installer.

We will also note the following important aspects that we are not the subject of this paper. (1) Such systems are not feasible without data security; there is a large literature on this subject. (2) Such systems often need to integrate heterogeneous data; here we follow established work on mediator-wrapper systems [27]. (3) Such systems are not feasible if sites do not have means of limiting the CPU and storage resources they share; various solutions exist, providing varying levels of security and privacy (4) Piracy risks in such systems, as in the Internet at large, raise a number of cyber-sociological and legal problems; we do not make this any of our business except to note that useful technology tends to find a way.

Contributions We do not intend this project to produce any substitutes for established work in the area of query optimization for distributed databases or in the area of database integration. In fact, we need and use some of that work whenever it applies to problems that arise in resource sharing (eg., metadata integration, cost-based distributed join optimization).

Instead, we design a distributed query language aimed specifically at solving the resource sharing problems outlined. We do this by proposing a set of process manipulation primitives that can be added to any query language. We also identify a small set of basic steps that a distributed *installer* for this language can follow, and we show that the resulting system is still quite expressive, covering many techniques such as brokering, proxying, etc. We provide a general algorithm for the installer that copes with the absence of ubiquitous and complete knowledge of resources, capabilities, and cost. We also show that it is possible to provide some static guarantees about the behavior of the installer.

Overview In Section 2 we review examples of resource sharing systems and previous work that can bear on the problems discussed in this paper. In Section 3 we describe the distributed process primitives that constitute our language illustrating them with a couple of simple examples and a more complex example involving a software distribution system. We also discuss the implementation of caching and subscriptions. The details of the installation mechanisms that perform some rewritings on query processes are presented in Section 4. If not handled carefully, installation is not guaranteed to terminate. We present an algorithm to prevent such cases in Section 5. We finally offer some future work and our conclusions.

2 Background and Related Work

Resource sharing has recently captured the interest and imagination of the masses with peer-to-peer file-exchange systems such as Napster[3], Gnutella[4] and FreeNet[5].

Even though peer-to-peer is an ancient network paradigm and has been already proposed as a database architecture [9], it appears to be used in a new environment, on a larger scale, to address new issues.

In recent resource sharing architectures, the sharing is not restricted to the data but also embraces processing and storage. Moreover, the data involved is not just of any kind. Most systems exemplify data defined as metadata + payload. The payload is usually a blob (a few megabytes) and the metadata is often semi-structured. Here are some examples.

Domain	Metadata	Payload	Example
music	title, genre, author, year	MP3 file ($\sim$ 4Mb)	Napster, Gnutella
movie	title, director, cast, year	divX file ($\sim$ 500Mb)	
software	package, distribution, architecture, version	RPM file (a few Mb)	

Napster and Gnutella

Napster is a mediated peer-to-peer system where users can advertise music files they are willing to share on Napster central server (this is done automatically when a user starts her Napster client and connects to Napster). The server maintains a real-time global catalog of available resources but does not store any music file. Users' requests sent to the server are answered in the form of a list of referrals. The user then picks the one(s) she wants and can download the resource directly from her peer.

Gnutella is *pure* peer-to-peer in the sense that it does not require a central server: there is no global catalog and users' requests are simply passed from peer-to-peer. A peer forwards the request to all the peers it knows. This can create some big network congestion.

These two systems (and their clones) offer very poor and inaccurate metadata and no query language. In Gnutella, the query is a string and can be interpreted differently by peers. Sharing is limited to read-only resources and none of them share processing. Moreover, query installation (how a query reaches the right host) is a critical weakness in Gnutella. Our framework addresses this issue specifically.

Distributed/Wide-Area Query Processing

The state of the art in distributed query processing is nicely presented in the recent survey [23] by Kossmann while the classic work in distributed databases is covered in [33]. The standard approach remains the one of System R* [29] with master and apprentice sites: the master site is in charge of developing the global plan and make the inter-site decisions; apprentice sites take the portion of the global plan that are relevant for them and develop a local plan for themselves, within the constraints of the global plan. Mariposa [37] presents a solution where every node is governed by economic motivations. The optimization remains centralized (a central optimizer decides join orders and methods) but the affectation of subtasks is based on bids and offers with negotiated prices, for a given budget. Systems interested in distributed query optimization also include Garlic [18], DISCO [39], DIMSUM [15] and ObjectGlobe [5]. We cannot directly apply this body of work because it relies on the full knowledge of the execution tree of a given query, while we assume very limited knowledge about it.

The special case of "*fusion queries*" [41] (where data is not cleanly fragmented like in traditional databases, and the number of participating sources is very large) is also relevant in our context. Some evaluation

[3] `http://www.napster.com`
[4] `http://gnutella.wego.com`
[5] `http://freenet.sourceforge.net`

strategies specific to client-server environments have been addressed in [14] (query/data/hybrid shipping) and [35] (code shipping). Relevant work on cost models includes total-cost estimates [30] and response-time estimates [16]. Cost models for wide area applications are addressed in [6].

Adaptive Query Processing

As pointed out in [21], the typical environment for data-integration (in the broad sense) implies the absence of reliable statistics (if any), the unpredictability of data arrival (initial setup and bursty arrival) and some overlap and redundancy among sources. It therefore requires query processing to adapt to these changing conditions. See [19] for a recent survey and comparisons based on the frequency of adaptivity.

In Tukwila [21] planning and execution are interleaved to make it possible to re-optimize queries if needed. Joins are performed using a variation of the double-pipelined join, operators are memory conscious and a special operator (collector) is used to handle the union of overlapping sources. In [3], adaptivity is pushed at the level of tuples.

Another aspect of adaptivity is caching (*"a by-product of query processing"* [17]). Query caching [2] and semantic caching [12] have already received a lot attention. The difficulties of integrating caching with query optimization have been addressed in [24].

Long-Lived Queries and Subscriptions

Wide-area applications have to offer efficient and scalable mechanisms to handle a large number of queries that process current and future data. The purpose of such queries is to monitor data sources and notify users when changes occur. Previous work on parallel database and query grouping [36] (aka multiple query optimization) is relevant here.

More recently the use of continuous queries [28, 11] has required the use new optimization techniques to handle both time- and content-based subscriptions. These frameworks also introduce a ad-hoc notion of *installation* to group queries to favor sharing and limit trigger overhead. As we will show, ours is more systematic and clearly defined. Our language can also represent the split operator from [11].

Push and pull mechanisms have been discussed in [15]. Relevant solutions can also be found in the "event distribution" literature (messaging and notification services) such as [4, 34] where efficient matching algorithms permit to scale a very large number of subscriptions and events. Our language naturally captures the *information flow graphs* from [4].

Brokering and Catalogs

Some of your routing strategies have been directly inspired from LDAP [20] and KQML [26].

Our advertising mechanism is similar to [27] and [13]. For the implementation of brokers, we can take advantage of broker architectures like *GlOSS* [38]. However, traditional data model (relational or object-oriented) do not seem to fit the chaotic nature of resource catalogs where metadata is partial and need to be enriched very often. Proposals like RDF [40] (*Resource Description Framework*) seem to specifically address some of these issues.

Mobile Computation and Software Agents

Software agent systems such as SIMS [22] are based on KQML [26] a communication layer that has "performatives" such as recruit, refer, etc. In SIMS/KQML however there is a clear separation between the performatives and the query language used (LOOM), and it is not clear how to implements several of the features we exploit.

The past few years have seen the development of a number of compact and elegant formalisms capturing various aspects of mobile agent computation most of them based on the pi-calculus [32]. Some process-based abstractions for global computations have been presented in [7] and the special case of combinators for Web services in [8]. Our choice of primitives to extend the query language has been directly inspired from this body of work.

3 A Distributed Query Language

We describe here a set of primitives for distributed processes that can be added on top of *any* query language (eg., SQL, Quilt [10]). For concreteness we use in the paper an OQL-like syntax, but our focus is on systems with sites, resources, channels, and processes.

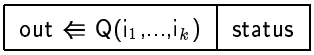
A system consists of several *sites*, located on various nodes of the network. A site is uniquely identified by its name. Each site stores some *resources* (data). Resources can be read and written only by processes executing locally. Resource names are locally unique. In the examples of this paper, we have two kinds of resources: metadata and "payloads". We can imagine, for example, that metadata is maintained by structured or semi-structured DBMS while payloads are plain files [6]. Payload and metadata are communicated between sites using named *channels*. Each channel is *located* at one site. Moreover, channels are created only by programs running at the site where the channels are located.

3.1 Query processes: installation and migration

At any site a number of *query processes* (abbrv. *qp*'s) can exist in parallel:

$$\boxed{P_1} \parallel \ldots \parallel \boxed{P_n}$$

To describe the *qp*'s themselves, we begin with the simple features of the language. In Section 3.3 we will add several more complex ideas. In simple form, a *qp* consists of an *output*, an expression (in some query language) containing several *inputs* (possibly none), and a *status*:



Inputs and outputs are located names (or expressions that evaluate to located names). We use Name@Site where Name can be (1) a channel name, (2) a resource name, or (3) an identifier for an (unmaterialized) view definition.

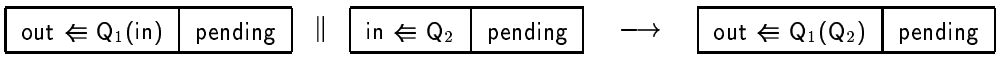
The status is a tag indicating one of the following

[execute]	Ready to read its inputs and write its output.
[pending]	Needs further *installation*.
[pending@S]	Scheduled for *migration* to remote site S, where its status becomes *[pending]*.

Query processes in *[execute]* status are then handled by an *execution manager* program at each site that actually runs them, using channels, reading and writing the data in the resources. Query processes in *[pending@S]* status come to the attention of another program, a *migration manager* that will move them to site *S*, in cooperation with the migration manager there.

¿From our perspective, the central component of the architecture is the *installer* program. The installer, the execution manager and the migration manager all run continuously and concurrently, daemon-style, at the site The installer examines the pending *qp*'s at the site and performs repeatedly the following *rewritings steps*:

Merge Two pending *qp*'s P_1, P_2 are replaced by one pending *qp*



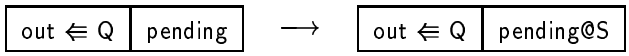
The rewriting applies only if there is a *match* between one of the inputs of P_1 and the output of P_2.

[6] However, large amounts of metadata are often grouped in "catalogs" which can in turn be seen as payload by some processing phases.

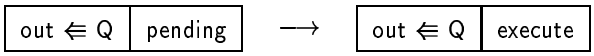
Split One pending *qp* P is replaced by two pending *qp*'s that would reconstruct P if merged. A *new* local channel or resource g is created and used as "glue":

The rewriting applies only if $Q_1(Q_2)$ is *equivalent* to Q.

Tag for migration The status of a pending *qp* is changed to *[pending@S]*.



Tag for execution The status of a pending *qp* is changed to *[execute]*.



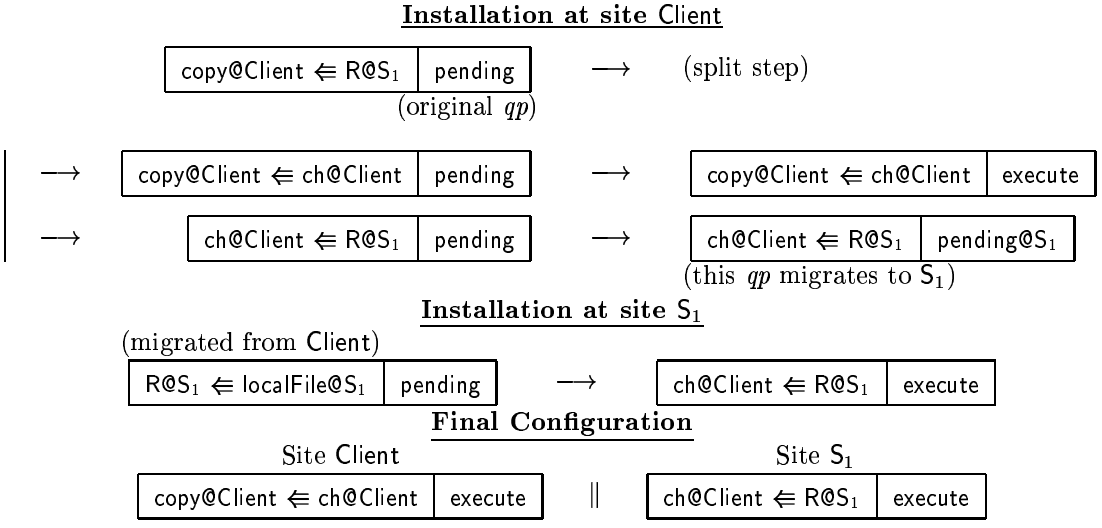
Clearly, the installer will apply the tagging steps subject to restrictions. For example, tagging for migration should apply only if the output is *not* a local resource. On the other hand, tagging for execution should apply only if *all* the inputs are local resources or local channels and if the output is a channel (remote or local) or a local resource. Additional constraints and installer strategies are discussed in Section 4.

For the moment we leave undefined the "match" in the merge step. Still, inputs and outputs that are the *same* located name should match under any definition. It is also understood that if we have a resource called `localName` at `Site` then the meaning of the expression localName@Site in a query is the the actual content of the resource (eg. content of a file).

The definition of "equivalent" in the split step will depend on the language and its semantics. Still, the "boundary" cases of splitting when $Q_1(x) = x$ and $Q_2 =$ in should be valid with any query language and arise very often. We call the first one *output splitting* and the second one *input splitting*.

3.2 Simple examples: shipping strategies

To better understand the primitives we just introduced, let us consider a simple but often encountered situation: a Client wants to copy from server S_1 a remote resource R. This is done by the following sequence of splitting, tagging, and migration steps: of



Only slightly more complicated are two standard strategies from distributed query processing: query-shipping and data-shipping. We assume three sites: Client and two servers S_1 and S_2. The client needs

the result of the join of relation R at S_1 with relation S at S_2 to be available on channel res locally. Both strategies are illustrated by Figure 1 and the corresponding query process is QP.

QP is defined as | res@Client ⇐ R@S_1 ⋈ S@S_2 | pending |

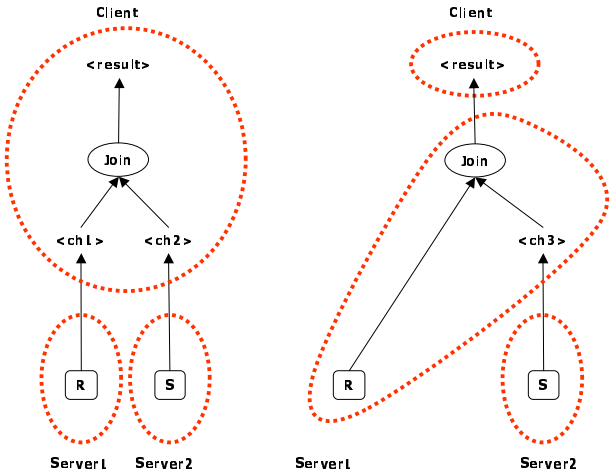


Figure 1: Shipping Strategies

For data-shipping (Figure 2), both remote inputs (R@S_1 and S@S_2) are replaced by two new local channels (ch_1@Client and ch_2@Client) and two new *qp*'s are created (as pending) then marked for migration to the remote servers.

QP	→	res@Client ⇐ ch_1@Client ⋈ ch_2@Client	pending	→	res@Client ⇐ ch_1@Client ⋈ ch_2@Client	execute
	→	ch_1@Client ⇐ R@S_1	pending	→	ch_1@Client ⇐ R@S_1	pending@S_1
	→	ch_2@Client ⇐ R@S_2	pending	→	ch_2@Client ⇐ S@S_2	pending@S_2

Figure 2: Data Shipping

For query-shipping (Figure 3), the initial *qp* is marked for migration to S_1. Once there, the remote input (S@S_2) is replaced by a local channel ch_3@S_1 and a new *qp* is created (as pending) then marked for migration to S_2.

QP → | res@Client ⇐ R@S_1 ⋈ S@S_2 | pending@S_1 |

The query process migrates to site S_1 and is *installed* locally.

→	res@Client ⇐ R@S_1 ⋈ ch_3@S_1	pending	→	res@Client ⇐ R@S_1 ⋈ ch_3@S_1	execute
→	ch_3@S_1 ⇐ S@S_2	pending@S_2	→	ch_3@S_1 ⇐ S@S_2	pending@S_2

(will migrate to S_2 for further installation)

Figure 3: Query Shipping

Three other common query routing mechanisms (inspired from both LDAP [20] and KQML [26]) are chaining, referral and recruiting. These involve a client site, a server site, and an intermediate site that

decides on the routing (see Figure 4 where there are multiple servers). These routings can be easily encoded in the same spirit as data-shipping and query-shipping: referral and recruiting involve possible merges followed by migration to client (referral) or server (recruiting). Chaining involves a split and a migration to server.

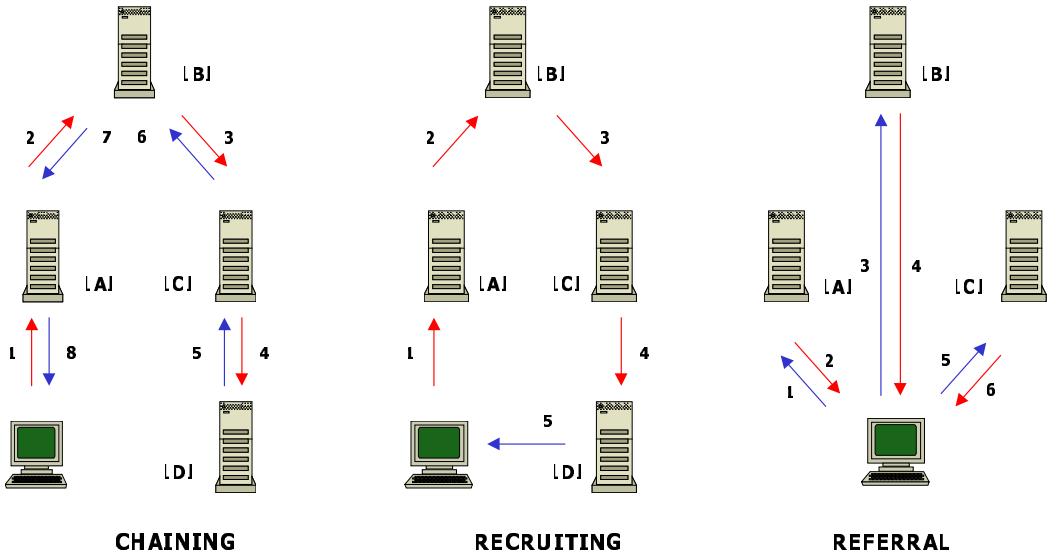


Figure 4: Query Routing Mechanisms

3.3 More complex features

Replicated *qp*'s For advertisement and subscriptions we will need *qp*'s with a *[replicated]* status. These are like `pending` *qp*'s, so the installer can merge and split them. The difference is that while a pending *qp* disappears in a merge, merging a replicated *qp* keeps a copy of it available for further installation:

out $\Leftarrow Q_1$(in) | pending	$\|\|$	in $\Leftarrow Q_2$ | replicated
$\hookrightarrow$ out $\Leftarrow Q_1(Q_2)$ | pending	$\|\|$	in $\Leftarrow Q_2$ | replicated

Note that the result of the merge itself is a pending *qp*. The result of merging two replicated *qp*'s is a replicated *qp* (and the original *qp* survive). A replicated *qp* is normally split into a replicated *qp* and a pending one. In special cases (subscriptions) the installer can decide to split it into two replicated *qp*'s.

We will also need corresponding tagging steps:

Tag for replicated migration
The status of a replicated *qp* is changed to *[replicated@S]*

out $\Leftarrow$ Q | replicated	$\rightarrow$	out $\Leftarrow$ Q | replicated@S

The *qp* migrates to site S where its status becomes *[replicated]*.

Query templates on outputs The merging of a pending *qp*, call it P, with another (typically replicated) *qp*, call it A will often, as we shall see, be used toward answering the query within P. We can think of A

as the *advertisement* of a capability. To allow for general formulations of such advertisements we extend the notion of output of a *qp*. In this more general form, a *qp* has the form:

$Q_0(X_1,\ldots,X_n) \Leftarrow Q_1(X_1,\ldots,X_n)$	replicated

$Q_0(X_1,\ldots,X_n)$ is a query *template*, i.e., containing metavariables $X_1,\ldots,X_n$ (Q_0,Q_1 also have usual inputs) that are used during merging just as with a parameterized rewrite rule. For example, the installer can perform the following merge step:

		$X@Sv_1 \bowtie foo@Sv_3 \Leftarrow X@Sv_2 \bowtie bar@Sv_2$	replicated
	$\|\|$	result@Client $\Leftarrow$ ick@$Sv_1 \bowtie$ foo@Sv_3	pending
$\hookrightarrow$		$X@Sv_1 \bowtie foo@Sv_3 \Leftarrow X@Sv_2 \bowtie bar@Sv_2$	replicated
	$\|\|$	result@Client $\Leftarrow$ ick@$Sv_2 \bowtie$ bar@Sv_3	pending

where X has been bound to ick.

Delayed queries Delaying query evaluation and spawning query processes during execution are two other additional features that play a crucial role in our treatment.

Imagine that an interested client will start with some metadata about a resource it wants. From this, perhaps with the help of some broker sites, using one or more *catalogs* of metadata [7], the client will "identify" the resource. But it is not clear that this always means a site name and the local name of a resource at that site. More flexibility is allowed if what is found, as a piece of metadata, is a *query*. The result of the evaluation of this query, call it G, should be the content of the desired resource.

To make this work, we need first to *not* evaluate G during the evaluation of queries that operate on the metadata that surrounds G. For this we use the syntax delay(G) and we modify the query evaluator to treat it as an inert value. This is what we call a *delayed query*.

Process spawning After receiving the text delay(G) as just (meta)data, the client needs to "activate" G. For this we extend the language by adding a *spawning* operator. The evaluation of spawn(out,delay(G),status) has the side-effect of creating a *qp* in which G is not delayed anymore:

eval(spawn(out, delay(G), pending) $\rightsquigarrow$ | out $\Leftarrow$ G | pending |

In order to establish precisely when in the query evaluation the spawning takes place, it will be convenient to add to the query language the following construct:

expr before *spawning*

With this, first *expr* is evaluated (which may create some bindings used in *spawning*) and then the spawning takes place.

Replicated execution The spawning feature allows us to consider for execution not just pending *qp*'s but replicated ones as well. To allow for "repeated execution" all we have to do is spawn the original *qp* replicated *qp* at the end of the execution:

Tag for replicated execution

out $\Leftarrow$ Q	replicated	$\longrightarrow$	out $\Leftarrow$ Q before spawn(out, delay(Q), replicated)	execute

[7] What constitutes metadata depends on what we do with it. For example, initially these catalogs can also be regarded as payload, described in turn by (meta!)metadata.

Installer decisions In what follows we will present several examples that involve complex sequences of installation steps. We discuss in Section 4 how the installer can choose between several rewrite steps that are applicable simultaneously. In the examples that follow we will present the behavior of the installer without reference to *how* decisions are made, simply saying "the installer will do this or that".

3.4 A more complex example

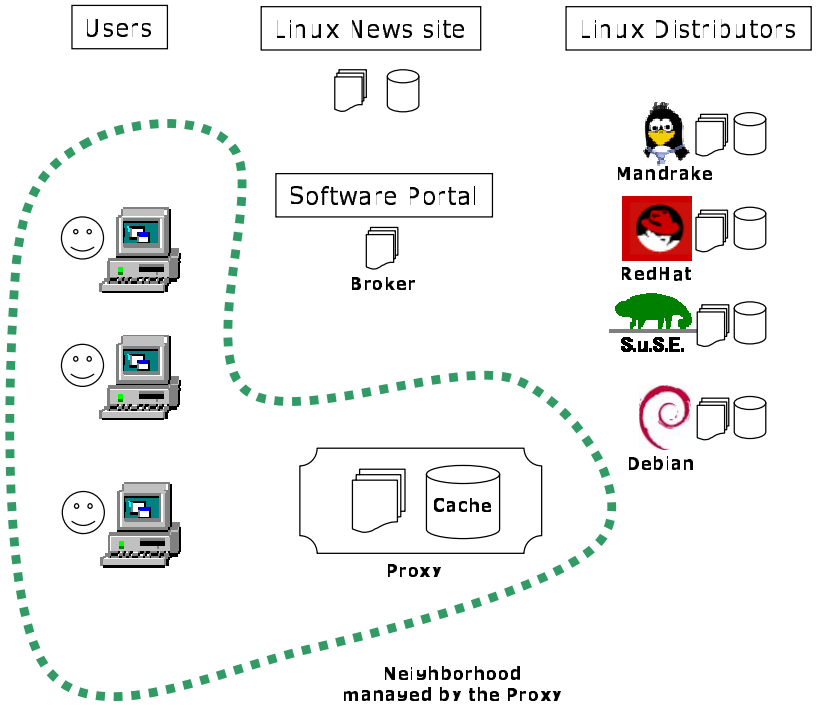


Figure 5: Main Example

We are now ready to present an example in software distribution. [8] Suppose that a Linux user wants to upgrade her `ssh`-related components. Let us also assume that a broker site B1 offers an up-to-date catalog (also a resource!) called News that contains information about the about available upgrades (advanced features, security fix, etc.) and how critical they are. This tells us *what* upgrade packages we need. To also find out *where* these upgrades are available, we will use another broker site B2 where the catalog Pkg pairs package names with resource access information. So far this is a simple problem, corresponding to an integration query such as:

Integration Query

```
select y.pkgname, y.version, y.whereItIs
from   x in News@B1,  y in Pkg@B2
where  x.priority >= 3 and  x.pkgname like "ssh"
  and  x.pkgname = y.pkgname and  x.latest = y.version
```

Now, instead of the user just reading the output and manually initiating file transfers, we want a file transfer to be spawned automatically for each `y` from `y.whereItIs`. We will show how our language supports this.

[8] An area in much need of improvement!

It is possible, even likely, that our user belongs to a *network neighborhood* where other users may have already done this particular upgrade. Therefore rather than just running all these file transfers from the user's site, we would like to make it the job of a proxy site in charge of Linux-related resources in this particular neighborhood. This proxy site may have cached some of the packages. Or it may recall which other users in the neighborhood recently retrieved these packages. In both cases, accessing such a "local" (close) copy can be cheaper than connecting to one of the Linux distribution site. We will also show how our language supports this behavior.

The example in our language The `whereItIs` attribute in the catalog Pkg@B2 is a delayed query of the form `delay(G)`. The way `G` tells how to get the corresponding package can, for example, be based on knowledge of a site and a local catalog at that site. The local name of the package can be obtained by querying the catalog. Let `G(loCa,Site,pN,v)` be (in OQL `element` extracts the unique elt of a singleton set):

G(lC,sS,pN,v) :=

```
define localN as select x.localName
                 from   x in loCat@Site
                 where  x.pkgName=pN and x.version=v
in  element(localN)@Site
```

(Recall that the meaning of `resName@Site` in a query is the the actual *content* of the resource `resName`.) We formulate the original query at the Client site as

Site: Client

```
localCatalog@Client ⇐
   select let f=newResource(Client) in tuple( y.pkgName, y.version, f)
          before spawn(f@Client, y.whereItIs, pending)
  from x in News@B1, y in Pkg@B2
  where x.priority > 3 and x.pkgName like "ssh"
    and x.pkgName = y.pkgName and x.latest = y.version       [ pending ]
```

The installer at Client decides to split this into a *qp* in charge of the spawning at Client and another in charge of integrating $\mathsf{News@B}_1$ and $\mathsf{Pkg@B}_2$. The glue between them is a newly created channel $\mathsf{c}_1\mathsf{@Client}$.

The installer also decides to tag the second *qp* for migration to site B_2.

Site: Client

```
localCatalog@Client ⇐
  select let f=newResource(Client) in tuple(z.pkgName, z.version, f)
         before spawn(f@Client, z.whereItIs, pending)
  from z in c1@Client  [ execute ]
```

(A)

Site: Client

```
c1@Client ⇐
  select tuple( y.pkgName, y.version, y.whereItIs)
    from x in News@B1, y in Pkg@B2
   where x.priority > 3 and x.pkgName like "ssh"
     and x.pkgName = y.pkgName and x.latest = y.version ) [ pending ]
```

Metadata integration For the metadata integration (*qp* with output $\mathsf{c}_1\mathsf{@Client}$), the installer decides to use data-shipping. The query (A) can be handled just like the example in Figure 1, modified to take into account the additional selections on $\mathsf{News@B}_1$ (just split the query between the join and the selections). After installation at Client, B_1, and B_2 we end up with the configuration of execute *qp*'s in Figure 6.

Site: Client

```
localCatalog@Client ⇐
( select let f=newResource(Client) in
  spawn(xy.whereItIs, f@Client, pending);
  tuple( xy.pkgName, xy.version, f)
  from xy in c1@Client ) [ execute ]
```

Site: B2

```
c1@Client ⇐
( select tuple( y.pkgName, y.version, y.whereItIs)
  from x in c2@B2, y in Pkg@B2
  where x.pkgName = y.pkgName
  and x.latest = y.version ) [ execute ]
```

Site: B1

```
c2@B2 ⇐ ( select * from x in News@B1 where x.priority > 3 and x.pkgName like "ssh" )  [ execute ]
```

Figure 6: Metadata integration: final configuration

Spawned *qp*'s The execution of the *qp* above at Client spawns query processes such as:

f001@Client ⇐ G(locPkg,RedHat,"ssh.client","2.1")	pending

Two important remarks.

First, the output of the spawned *qp*'s is different for each of them. The newResource() method creates a fresh new resource name every time it is called. In the example, we represent these fresh new names using f001, f002, etc.

Second, every time a box is spawned the corresponding resource name gets added to localCatalog@Client.

(B)

f001@Client ⇐ G(locPkg,RedHat,"ssh.client","2.1")	pending

Using a proxy We now focus on the installation of a spawned *qp* (B). The obvious thing to do for the installer is to create a new channel name c, and create two new *qp*'s: (1) one that migrates to RedHat and asks for the content of the package to be sent on channel c; (2) one local that reads from channel c and writes into resource/file f001.

However, it happens that the Client can take advantage of the services of a proxy. The proxy – among other things – may maintain local caches (or even subscriptions!) of resources that are of interest to clients from its neighborhood, for example because they were recently downloaded. Even if the resource of interest is not cached, the proxy may be aware of what other clients from the neighborhood have recently downloaded it and are willing to share. In our case the proxy is in charge of the neighborhood's interactions with locPkg@RedHat.

The local installer is aware of the presence of the proxy because the installer there has advertised its services, migrating to the Client a *qp* with a query template in output:

G(locPkg,RedHat,Y,Z) ⇐ G(Pkg,Proxy,Y,Z)	replicated

Our *qp* (B) gets merged with this "advertisement" producing

f001@Client ⇐ G(Pkg,Proxy,"ssh.client","2.1")	pending

After output splitting with a new channel c55 and tagging for migration:

f001@Client ⇐ c55@Client	execute	‖	c55@Client ⇐ G(Pkg,Proxy,"ssh.client","2.1")	Pending@Proxy

What happens to the second *qp* after it migrates to the proxy?

Case 1: local copy
The proxy happens to have a local copy of the resource (say cache.ae45fe30). This can exploited with the presence at the Proxy of the advertisement

G(Pkg,Proxy,"ssh.client","2.1") ⇐ cache.ae45fe30@Proxy	replicated

After a merge, we have the executable configuration:

c55@Client ⇐ cache.ae45fe30@Proxy	execute	‖	f001@Client ⇐ c55@Client	execute

Case 2: copy in the neighborhood
The proxy happens to know a neighbor (Client_2) willing to share this resource (the name of the resource exported by Client_2 is cache.ed45f001). Again, we could have the advertisement:

G(Pkg,Proxy,"ssh.client","2.1") ⇐ cache.ed45f001@Client2	replicated

After merging, the proxy can decide to retrieve the data itself, store it locally in its cache and then send it to Client (chaining). It can also simply migrate the *qp* back to Client, letting it take care of the retrieving (referral). Or it can ask Client_2 to send the data directly to the user (recruiting).

Case 3: remote access
The proxy has no choice but to get the resource remotely. If none of the cases above are satisfied, the proxy needs to ask for the resource from a remote site. However, the proxy may know of some redundancy among servers or portals, having for example *two* advertisements:

G(Pkg,Proxy,"ssh.client","2.1") ⇐ G(locPkg,RedHat,"ssh.client","2.1")	replicated

G(Pkg,Proxy,"ssh.client","2.1") ⇐ G(PkgLoc,Mandrake,"ssh.client","2.1")	replicated

Next, the proxy may again choose among chaining (with caching), referral, or recruiting. At the remote server assuming the presence of mirror sites, the same game can be played, the remote site playing the role of a proxy for the proxy itself.

3.5 Caching and subscription

We have mentioned caching and even subscription as an activity that a proxy site may perform. The two are actually related: we can think of caching as a subscription with just one refresh, and of subscriptions as continuously refreshed caches. Here we show how both can be achieved with the language primitives presented earlier.

Assume that we are at a generic site called Client and the installer there makes the decision to cache data@Server. data may be an actual resource local to Server or it may be the name of a view defined at Server. The concrete action taken by the installer is to add to the set of *qp*'s the following one

data@Server ⇐ data@Server	replicated

This "cache" *qp* will not affect existing *qp*'s. None of them will merge with it because it does not change anything. However, in the next step the installer will split it by creating the local resource in which the caching will take place. This yields

(1)

data@Server ⇐ cache@Client	replicated

(2)

cache@Client ⇐ data@Server	pending

The *qp* (1) will serve in the future as the *advertisement* of the cache (see in Section 3.4 how the installer at Proxy used the advertisement of a cache). The *qp* (2) will then be installed as in resource copying in Section 3.2.

The implementation of subscriptions is more complicated. Once it has decided to subscribe to data@Server and it has determined what should trigger the refreshing of the subscription, the installer at Client will add the following *qp*.

(3) | data@Server ⇐ subscr(data@Server, trigger@Server) | replicated |

Here subscr is just syntactic sugar for an operation in the query language that evaluates (in execution) like the first projection of a pair: $eval(\mathsf{subscr}(\mathsf{E}_1,\mathsf{E}_2)) = eval(\mathsf{E}_1)$

Thus, trigger@Server does not play any role during execution. trigger is a view name whose role is to block the *qp* from entering execution status until another *qp* that defines trigger appears at Server. By using different trigger names (even templates with variables) we can represent various kinds of subscription (see discussion later). The installation of (3) proceeds as follows:

(4) | data@Server ⇐ dataLocal@Client | replicated |

(5) | dataLocal@Client ⇐ subscr(data@Server, trigger@Server) | replicated |

(4) is the advertisement while (5) splits further into

(6) | dataLocal@Client ⇐ chan@Client | replicated |

(7) | chan@Client ⇐ subscr(data@Server, trigger@Server) | replicated |

After tagging for replicated execution and replicated migration (see Section 3.3) we have

dataLocal@Client ⇐ chan@Client before spawn(dataLocal@Client, delay(chan@Client), replicated)	execute

chan@Client ⇐ subscr(data@Server, trigger@Server)	replicated@Server

At the Server site, *each time* a *qp* of the form | trigger@Server ⇐ dummy | pending | appears, it merges with

(8) | chan@Client ⇐ subscr(data@Server, trigger@Server) | replicated |

yielding | chan@Client ⇐ subscr(data@Server, dummy) | pending | .

This eventually sends the content of data@Server through chan@Client into dataLocal@Client, while a copy of (8) is kept and used in future refreshings.

More about triggers Without loss of generality, we can always distinguish between content-based and time-based triggers (see [28] and [11]). For time-based triggers, we can imagine that the local installer will create new *qp* at fixed times, like the Unix `crontab` daemon. Content-based triggers can be implemented by database triggers. The details are actually not that simple because we have to make sure that: (1) the installer generates enough *qp*'s for every *qp* waiting to be merged and (2) every waiting *qp* is merged with its trigger only once.

One solution is for the installer to maintain a catalog of *registered listener* for some triggers. Whenever the trigger is active, all the listeners are notified. We now impose a special installation procedure for trigger *qp*'s (this is implementation dependent).

We have just presented one way to represent triggers in our framework. This encoding is an *installation-time encoding* because the trigger is going to be used by the installer to perform some rewriting. We can also represent triggers using an *execution-time encoding*[9] where triggers are channels used as a guard for expressions: a given *qp* will be blocked on the channel until some information is sent through. Our encoding does not specify exactly how these triggered are fired because triggers are implementation and query language specific.

[9] We do not present the encoding here for lack of space.

Daemon for caching and subscriptions In the description above we have omitted certain features for the sake of simplicity. In fact it should be the installer itself that makes the decision to cache or subscribe to some resource. This is done by a separate daemon that monitors the behavior of the installer and logs statistics about *qp*'s in order to identify candidates for caching or subscription. These decisions are based on local policies and the current status of the node (memory/storage available) [2, 24].

3.6 Advertising

We have already mentioned a couple of times the notion of *advertising*, when we said that Napster users advertise their music files (Section 2) or when we compared query templates to advertisement of a capability (see Section 3.3).

We now define more precisely what we mean by advertising. First the notion is always related to a replicated *qp* because an advertising has to be used many times. Local advertisements correspond to replicated *qp*'s while remote ones correspond to replicated@remote (migration to the remote site, when the *qp* is then marked as replicated.

What makes an advertisement different from another pending (or replicated) process is that it has been created for some data and not for some query. Most pending and replicated *qp*'s are created via merge and split starting from an originating query. The reason of an advertisement is the presence of some data locally and the will of the installer to share this new resource either locally or remotely.

3.7 Leasing

As we have defined them, replicated *qp*'s never die. This is not a realistic assumption because some sites will shutdown or cease to provide some resources. One convenient way to solve the problem is to assume that replicated *qp*'s (advertisements, caches, subscriptions, etc.) are defined with a lease. The termination of the lease implies the termination of the query process itself.

The interesting aspects of leasing are that: (1) it is an intuitive concept; (2) it fits both client and server concerns; (3) it solves some distributed computing problems such as resource management; and (4) is being already used for distributed architectures and therefore is available in the form of software components.

We can imagine that the details of the leasing contract will be negociated by installers. Also, knowing that a leasing contract holds, a site can and sometimes must use optimizations that rely on the contract itself.

We can also think of extended use of leasing, to capture the following aspects:
Freshness of information: Leasing can be very useful when dealing with metadata or catalogue information. If each metadata item is captured through a lease, it means that the item (i.e. the informatin) will be valid as long as the lease holds. Using lease is a way to force the entities involved in the network to refresh their information. The same principle can be applied to data itself, especially for data with a high variability such as real-time data.

Garbage collection: In a distributed network where resources are scarce, it is crucial to have a good management of them via garbage collection. With mechanisms such as subscription of remote cache that can consume a lot of server resources, it is important to make sure that allocated resources are effectively being used. Leasing is an intuitive way to manage such resources: as long as a client expresses interest in a resource, the server will keep the resource available; if interest is not renewed, the server can use the resource for something else.

4 Installation strategies

As already mentioned installation can be seen as a local daemon that rewrites query processes marked either as pending or replicated. We do not describe exactly in which order the installer will process these *qp*'s but we can imagine *qp*'s organized in circular list (one *qp* handled at a time) or something more complicated where *qp*'s are grouped by input, output, shapes, etc. This is left to the implementation. The main actions taken by the installer are based on four rewriting primitives: merge, split, tag for migration and tag for execution (see Section 3).

The main difficulty in designing installation strategies is the fact that it is not possible for each site to have cost information about *all* other sites. Usually, the metadata in resource sharing systems includes some of this information. If so, this can be made available to installers through various mechanisms. When we do not have cost information we use heuristics based on the following principles:

- If a *qp* present at some site mentions local view definitions we should first resolve (expand) those, by merging.
- If a *qp* present at some site mentions local resources (payload or catalog) it is better to process locally the subqueries involving those resources.
- If we have a *qp* with remote inputs at several sites, and we do not have cost information about catalog or payload sizes, the installer can obtain dynamically network information using *fake requests* (see also below) and use it in choosing the site where the *qp* is to migrate.

Installation is needed mostly by pending *qp*'s. We omit a detailed discussion of the installation of replicated *qp*'s referring to our discussion of advertising (Section 3.6) and subscription (Section 3.5).

An installation algorithm We present an installation algorithm consisting of three phases. The phases refer to installation steps performed on the same "current" *qp*. However, since *qp*'s are processed according to a schedule, the installer will give attention to each *qp*'s for a fixed period of time, in a round-robin fashion. A less fair alternative is to complete an entire phase for each *qp* before switching to another. The shape of the current *qp* dictactes which phase of the installation needs to be performed. We explain the three phases individually before we present the algorithm.

Phase 1:

We apply merge steps to the current *qp* until *all* inputs that are local view definitions are resolved. In general, merge steps can be applied in more than one way, both because there may be several local view names among the inputs and because they may merge with more than one other *qp* (recall installation at Proxy in section 3.4). Therefore, in this phase we need to explore a search space, prune it and come up with the best solution using cost information or heuristics We postpone a discussion of this for the end of this section.

At the end of Phase 1, the *qp* is has the following form

$\text{out} \Leftarrow Q(\text{local}_{IN}, \text{remote}_{IN})$	pending

where local_{IN} is the set of its local inputs, all of which are either local resources or local channels, and where remote_{IN} is the set of its remote inputs.

Phase 2:

We apply two split steps resulting in

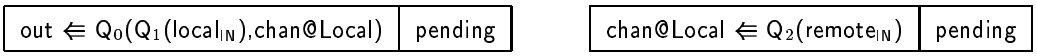
$\text{out} \Leftarrow Q_0(Q_1(\text{local}_{IN}), Q_2(\text{remote}_{IN}))$	pending

This can be done using standard but language-dependent query decomposition techniques [33] that separate the processing in the most advantageous way.

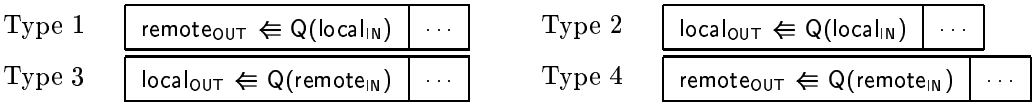
The next step is to decide using cost information (if available) whether it is better to evaluate Q_0 locally or remotely. This decision can be made using distributed query optimization techniques (eg., the iterative dynamic programming algorithm in [25]).

If remote evaluation is better, then the optimization algorithm used will also identify a remote site where the query should be further installed, and the installer here tags the entire *qp* for migration there. There is no Phase 3 in this case.

If local evaluation is better, or if not enough cost information is available then the installer splits the *qp* using a local channel:



Of course, if the *qp* does not have a mixture of local and remote inputs, Phase 2 does nothing. In any case At the end of Phase 2, the *qp* is of one the following four types:



Phase 3:

For types 1 and 2, the installer simply needs to tag the *qp* for execution: all the inputs are either local channels or local resources.

For type 3, we are in a situation that existed also in Phase 2. If we have enough cost information we can again use a distributed query optimization algorithm to choose the site where the *qp* should migrate. The installer splits the query to implement query-shipping (see Section 3.2) bringing only the result back. One alternative that the optimization algorithm will consider is to evaluate the *qp* locally, since the result is needed here. If this solution is picked, the installer will use further split steps to implement data-shipping (see Section 3.2) bringing the remote data to the local site. If we do not have enough cost information we choose query-shipping, using the last heuristic listed above to decide to what site to send the query.

Type 4 queries are handled like type 3 except that no solution involves local processing. If the cost-based approach is possible and suggests migration to remoteout, this is implemented as *referral* otherwise as a *recruiting* (see Section 3.2) of whatever site comes out of cost-based optimization. If we do not have enough cost information we do recruiting relying again on the last heuristic listed above. (Additional heuristics suggesting referral can be considered.)

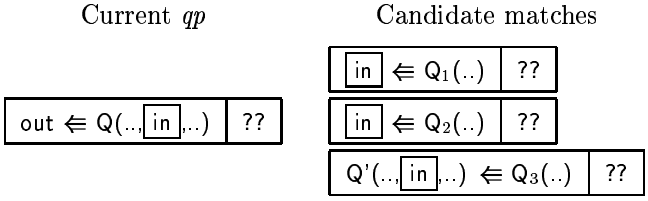
The installation algorithm can be specified as follows:

Installation Algorithm

```
1   while (true)
2    pick the next qp
3     while (allocated period for qp not over)
4       repeat
5         if qp contains local views (as input)
6           then apply Phase 1
7         if qp is not of types 1, 2, 3 or 4
8           then apply Phase 2
9         if qp is of types 1, 2, 3 or 4
10          then apply Phase 3
11          else continue
```

Note first the installer allocates a period for each *qp*. The period can be measured as a time or as a number of rewritings performed. When none of the cases can be appplied (line 11), we simply move to the next *qp*.

Searching the space of merging alternatives When installing the current *qp*, we apply the merge rule by picking one input from the *qp* (say in) and identifying some matching candidates where in appears as an output or template output, as illustrated below.



The same process is repeated for the candidates, the candidates of the candidates and so on. The resulting search space can be represented as a tree where the root corresponds to the current *qp* being rewritten by the installer.

In any case, the decision (because at the end we have to end up with one) will be based on costing among various candidates. Costs will be used to compare candidates and also to prune the exploration tree.

Cost model In choosing between several merge steps we look at the inputs that are brought in by the merges, since the output of the resulting *qp*'s is the same. The inputs added in different merges represent different ways of obtaining a resource. Let us illustrate with an example where we ignore the cost of local processing and we focus on the transmission cost (TC) (This is quite realistic unless we are joining metadata from large catalogs.)

A site may have a certain resource either cached locally, or available from a peer in the neighborhood or needs to retrieve it from a remote site (the source). Each of these is represented by an advertisement *qp* that can merge with the *qp* that needs the resource. (Recall the proxy for Linux packages in Section 3.4.)

Each of these three scenario have a cost TC_{cache}, TC_{peer} and TC_{source}. We assume that the transmission costs from peers in the same neighborhood are the same and that

$$TC_{cache} < TC_{peer} \ll TC_{source}$$

We also have a probability distribution p_{cache} (resource cached), p_{peer} (resource at peer but not cached) and p_{source} (resource available from the source only), with

$$p_{cache} + p_{peer} + p_{source} = 1$$

We can now assign a cost to a given candidate:

$$C = p_{cache} * TC_{cache} + p_{peer} * TC_{peer} + p_{source} * TC_{source}$$

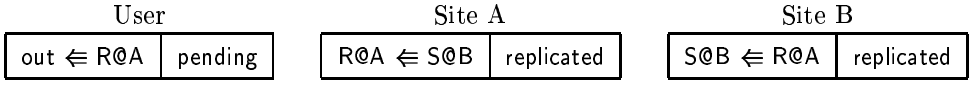
For a given set of resources (like Linux packages in our example), the TC are fixed (probably parameterized depending on the time of the day. See [6]) and the only information that need to be updated frequently are the probabilities. This information could be piggy-backed onto advertisements migrated to the client.

Of course, we may also end up with incomparable alternatives because of lack of cost information. In that case, we use heuristics to force a decision. For example, we may need to compare the cost of accessing two remote resources through portals, without knowing the mirroring structure behind the portals. (Recall our example, with RedHat and Mandrake.) In such cases, we evaluate the cost by sending *fake* requests[10] to these sites, in the spirit of the `ping`[11] and `traceroute` Unix commands. Since we assume that every node of the network is running our client, we can have this service made available.

[10] The HTTP protocol offers the `HEAD` request to get some information about a page without downloading its content.
[11] `ping` is used by Napster to evaluate the closeness of a peer.

5 Detecting Cycles in Installation

The installation algorithm can cycle, in fact on a quite simple example:



The user *qp* will migrate to Site A, to be merged with R@A. The resulting *qp* will then migrate to Site B (recruiting), to be merged with S@B. The resulting *qp* will migrate back to Site A, and so on. This is a recruiting strategy, but chaining and referral would create the same cycles and would require some extra channel names and boxes.

Cycles are a problem because they make the installation not terminate. As we shall see, cycles can originate from mutually recursive view definitions or naive installation decisions. In this section we address these two issues. For the first one we provide a detection algorithm that works on definitions. For the second issue, we simply prove that assuming no cycles in view definitions, our installation algorithm itself will introduce no cycles

To test for cycles in view definitions we have several options:

Off-line detection Before starting any query, we run an offline detection algorithm that looks for cycles. This might be expensive and too restrictive, because most running queries might not actually use these bad view definitions.

On-line detection Instead of running an off-line algorithm, whenever a new query comes in, we check during its installation that the view definitions it uses are not involved in a cycle. This might involve some duplicate work for similar queries. Even if we cache some per-query information locally, other nodes with similar queries will perform the same work.

Advertising-time detection The cycle-checking algorithm is now performed whenever a new view definition is advertised by the system. For instance when site A decides to advertise R@A, site A will have to check that the remote view definitions mentioned in the definition of R do not mention R itself.

Our approach is the third one: to run a checking algorithm whenever a new view definition is advertised or updated (i.e. re-advertised with the same name but different query body) to the system.

We have to solve two sub-problems. First we need to design an algorithm that tells whether or not an definition update is valid (i.e. it does not create cycles). We also need to make sure that concurrent definition updates are not messing with each other. We can imagine that site_1 is willing to update one of its definition while site_2 is willing to do the same. If both checks are made with before-the-update definitions, we might have some problems.

Cycle Detection Algorithm

```
Checks if site s can advertise D@s
input: D, timeout
output: yes/no/try-later
1. generate a timestamp ts and mark D@s
2. for every view definition R@s' in the definition of D@s
     send query Q(s,ts,R@s') to site s'
3. wait for the answers or timeout
4. unmark D@s
```

Queries sent during the execution of the algorithm have to be interpreted and answered in the following way.

- if site s' receives a request for $R@s'$ with timestamp ts and $R@s'$ is already marked with timestamp ts' such that $ts' < ts$, then site s' sends back a `try-later` answer.
- if site s' receives a request about $R@s'$ and R is an atomic view definition (does not depend on anything else), then it sends back an acknowldegment.
- otherwise for each view definition involved in the definition of $R@s'$ site s' sends a query and waits for the corresponding acknowledgment.
- if the site s' has received an acknowledgement for each of the sub-queries, it sends an acknowledgment to the sender of the query.
- if the site s' receives a `try-later`, it forwards it to the sender.

The termination of the algorithm is defined by the three rules:

- if site s receives a query about the very view definition it tries to advertise, then it knows there is a cycle. The update of $D@s$ is canceled.
- if site s receives a `try-later`, the update of $D@s$ is canceled. The algorithm can be tried later on.
- if site s receives an acknowledgement for each query it has sent, then the update definition is safe.

We want to make the following remarks. Compared to [1], we do not need to keep track of which requests have already been answered because we start from an acyclic graph. On the other hand, we have to take care of timestamps to prevent concurrent updates to interfere. We also have three possible outcome: cycle, no cycle and `try-later`. In the case of a cycle or a `try-later`, a lot of nodes might be waiting for an acknowledgement that will never arrive. Since the detection of the cycle is always made by the originating site s – who now knows there is a cycle –, we can always decide that s will send back an acknowledgement – that will bubble up back to s itself – and simply ignore it. We could also decide that nodes can timeout.

We now focus on the installation in general, assuming no cycles in view definitions. To state our safety result about our installation algorithm we need to associate with each site and each pending *qp* a *sequence of output channels* (SOC). Note that for a *qp* to migrate its output must be a channel. We assume that when a *qp* migrates it takes its SOC from its old site, appends its output channel to it and this becomes its SOC at the new site.

Theorem Given initial configuration of resources and (non-recursive) views, we can compute an upper bound on the length of any SOC (sequence of output channels) that can occur during subsequent installation with our algorithm.

6 Conclusion and Future Work

In this paper we have presented a way to extend a query language into a distributed query language with only a few new primitives. Queries are encapsulated into query processes. Query processes are installed by local programs sitting at each node of the network. During installation, query processes can combine with each other and migrate from sites to sites. Once installed, query processes are executed and can exchange data along channels. We have shown how traditional distributed query mechanisms (query/data/hybrid shipping, referral, chaining, recruiting) can be easily encoded in this new framework. We have also proposed algorithms to perform query installation in a systematic and bounded way. Installation decisions are based on cost estimates.

Our motivation for this work is grounded in the emergence of resource sharing environments that have already shown some serious shortcomings in terms of extensibility and scalability. Even though such environments show certain similarities with traditional database architectures, they also offer some specificities and optimization opportunities that do not seem to be handled properly by current

approaches. There is a clear distinction between metadata and payload both in terms of structure and size. Resources are organized in catalogs for which traditional join methods are not always adequate. Metadata often contains not only information about the payload but also a query to retrieve it and therefore requires some delayed evaluation.

The work presented in this paper is on-going research and a prototype of a distributed query language based on the Quilt proposal is under development. Our immediate concern is to demonstrate that such system can be built and deployed easily and efficiently. This will permit to validate our installation designs: even with limited knowledge, a local installer can make a locally good decision that will produce a globally good plan. The fine tuning and refinement of our cost model will be critical here.

Our language seems suitable for simulating traditional information architectures such as distributed databases or messaging services. More specifically we plan to investigate efficient ways to implement database operations along channels using stream-based algorithms.

Back to the language itself, the addition of new operators in the spirit of *service combinators* [8] seems to be really promising: they would permit for instance to express concurrency (i.e. competition) between query processes. But we have to understand how they interact (or interfere) with our other primitives at both installation and execution time.

References

[1] Serge Abiteboul and Victor Vianu. Regular path queries with constraints. In *ACM PODS*, 1997.

[2] S. Adali, K. Selcuk Candan, Y. Papakonstantinou, and V.S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *ACM SIGMOD*, pages 137 – 148, 1996.

[3] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *ACM SIGMOD*, May 2000.

[4] Guruduth Banavar, Marc Kaplan, Kelly Shaw, Robert E. Strom, Daniel C. Sturman, and Wei Tao. Information flow based event distribution middleware. In *Middleware Workshop at the International at the Conference on Distributed Computing Systems 1999*, 1999.

[5] Reinhard Braumandl, Markus Keidl, Alfons Kemper, Alexander Kreutz, Stefan Prls, Stefan Seltzsam, and Konrad Stocker. ObjectGlobe: Ubiquitous Query Processing on the Internet. Technical report, Universität Passau, 1999.

[6] Laura Bright, Louiqa Raschid, Vladimir Zadorozhny, and Tao Zhan. Learning Response Times for WebSources: A Comparison of a Web Prediction Tool (WebPT) and a Neural Network. In *CoopIS*, September 1999.

[7] Luca Cardelli. Abstractions for Mobile Computation. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *LNCS*. Springer, 1999.

[8] Luca Cardelli and Rowan Davies. Service combinators for web computing. *IEEE Transactions on Software Engineering*, 25(3):309–316, May/June 1999.

[9] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *ACM SIGMOD*, 1994.

[10] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Invited paper, WebDB-2000*, May 2000.

[11] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagraCQ: A Scalable Continuous Query System for Internet Databases. In *ACM SIGMOD*, 2000.

[12] Shaul Dar, Michael J. Franklin, Björn THór Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *VLDB*, 1996.

[13] S. Embury, J. Shao, W. Gray, and N. Fishlock. Advertising Database Capabilities for Information Sharing. In *CAiSE*, 2000.

[14] Michael J. Franklin, Björn THór Jónsson, and Donald Kossmann. Performance tradeoffs for client-server query processing. In H. V. Jagadish and Inderpal Singh Mumick, editors, *ACM SIGMOD*, 1996.

[15] Michael J. Franklin and Stanley B. Zdonik. "Data In Your Face": Push Technology in Perspective. In Laura M. Haas and Ashutosh Tiwary, editors, *ACM SIGMOD*, 1998.

[16] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query optimization for parallel execution. In Michael Stonebraker, editor, *ACM SIGMOD*, 1992.

[17] Laura M. Haas, Donald Kossmann, and Ioana Ursu. Loading a cache with query results. In *VLDB*. Morgan Kaufmann, 1999.

[18] Laura M. Haas, Renée J. Miller, B. Niswonger, Peter M. Schwarz Mary Tork Roth, and Edward L. Wimmers. Transforming Heterogeneous Data with Database Middleware: Beyond Integration. *IEEE Data Engineering Bulletin*, 22(1):31–36, 1999.

[19] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Sam Madden, and Vijayshankar. Adaptive Query Processing: Technology in Evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, June 2000.

[20] Tim Howes, Mark C. Smith, Gordon S. Good, and Timothy A. Howes. *Understanding and Deploying LDAP Directory Services*. MacMillan, Jan 1999.

[21] Zachary Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Dan Weld. An Adaptive Query Execution Engine for Data Integration. In *ACM SIGMOD*, 1999.

[22] Craig A. Knooblock and Jose Luis Ambite. Agents for information gathering. In Jeffrey M. Bradshaw, editor, *Software Agents*, chapter 16. MIT Press, 1997.

[23] Donald Kossmann. The State of the Art in Distributed Query Processing . Submitted to ACM Computing Surveys.

[24] Donald Kossmann, Michael J. Franklin, and Gerhard Drasch. Cache Investment: Integrating Query Optimization and Dynamic Data Placement. *ACM TODS*, December 2000.

[25] Donald Kossmann and Konrad Stocker. Iterative Dynamic Programming: A New Class of Query Optimization Algorithms. *ACM TODS*, March 2000.

[26] Yannis Labrou and Tim Finin. A Proposal for a new KQML Specification. Technical Report TR CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, MD 21250, February 1997.

[27] Chen Li, Ramana Yerneni, Vasilis Vassalos, Hector Garcia-Molina, Yannis Papakonstantinou, Jeffrey D. Ullman, and Murty Valiveti. Capability Based Mediation in TSIMMIS. In *ACM SIGMOD*, 1998.

[28] L. Liu, C. Pu, and W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Transactions on Knowledge and Data Engineering*, 1999.

[29] Guy M. Lohman, Dean Daniels, Laura M. Haas, Ruth Kistler, and Patricia G. Selinger. Optimization of nested queries in a distributed relational database. In *VLDB*, 1984.

[30] Lothar F. Mackert and Guy M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *VLDB*, 1986.

[31] David Maier. A Petabyte in Your Pocket: Directions for Net Data Management, March 2000.

[32] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[33] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 2 edition, 1999.

[34] Radu Preotiuc-Pietro, Joao Pereira, Francois Llirbat, Francoise Fabret, Kenneth Ross, and Dennis Shasha. Publish/Subscribe on the Web at Extreme Speed. In *VLDB*, September 2000.

[35] Manuel Rodriguez-Martinez and Nick Roussopoulos. Mocha: A self-extensible database middleware system for distributed data sources. In *ACM SIGMOD*, volume 29, pages 213–224. ACM, 2000.

[36] Timos K. Sellis. Global query optimization. In Carlo Zaniolo, editor, *ACM SIGMOD*, 1986.

[37] M. Stonebraker, P. M. Aoki, R. Devine, W. Litwin, and M. Olson. Mariposa: A New Architecture for Distributed Data. In *ICDE*. IEEE Computer Society Press, February 1994.

[38] A. Tomasic, L. Gravano, C. Lue, P. Schwarz, and L. Haas. Data structures for efficient broker implementation. *TOIS*, 15(3), July 1997.

[39] Anthony Tomasic, Louiqa Raschid, and Patrick Valduriez. Scaling Heterogeneous Databases and the Design of Disco. In *ICDCS*, 1996.

[40] W3C. Resource Description Framework (RDF) Model and Syntax Specification, February 1999. Available from `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222`.

[41] Ramana Yerneni, Yannis Papakonstantinou, Serge Abiteboul, and Hector Garcia-Molina. Fusion Queries over Internet Databases. In *EDBT*, 1998.